

Using MMX™ Instructions to Implement a 1/3T Equalizer

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004
* Other names and brands may be claimed as the property of others.

CONTENTS

1.0. INTRODUCTION
2.0. 1/3T EQUALIZER FUNCTION
3.0. IMPLEMENTING A 1/3T EQUALIZER IN MMX™ INSTRUCTIONS
 3.1. Reformatting the Input and Output Data
 3.2. Calculating the Filter Output Y(IQ)
 3.3. Formatting the Equalizer Output and Generating the Error Term
 3.4. Performing the Adaptation and Generating New Filter Coefficients
4.0. MMX™ TECHNOLOGY CODE PERFORMANCE
APPENDIX A. C-CODE WRAPPER FOR 1/3T EQUALIZER
APPENDIX B. MMX™ CODE FOR 1/3T EQUALIZER

1.0. INTRODUCTION

The media extensions to the Intel Architecture (IA) instruction set include single-instruction, multiple-data (SIMD) instructions. This application note presents examples of code that exploit these instructions. The 1/3T Equalizer algorithm presented here illustrates how to use the new unpack and multiply-add instructions (PUNPCKLDQ and PMADDWD) to perform an efficient complex multiply. This example demonstrates just one of the many ways that a programmer can take advantage of the MMX instruction set to obtain a performance gain over scalar IA code.

2.0. 1/3T EQUALIZER FUNCTION

One goal of a data communication system is to ensure that a transmitted data sequence is received the same way it is transmitted. A telephone presents a clear example of this. Namely, the received signal (spoken words) are heard at the receiver the same way the words are spoken at the transmitter end. In a data communication system, an equalizer function is used at the receiver end of a system to ensure that a transmitted data sequence is received the same way it is transmitted. Consequently, an equalizer function is found in almost all commercial modems.

The 1/3T Equalizer described in this application note is defined mathematically in Table 1.

Table 1. 1/3T EqualizerMathematical Definition

Element	Definition
Input:	x, complex, 16 bit precision
Equalizer coefficients:	h(n), complex, 16 bit precision
Output:	y, complex
Reference data:	d, complex
μ:	adaptation size
Equalizer output:	$y(i) = \sum_{n=0}^{L-1} x(3i+n)h(n)$
Coefficient adaptation:	$h(n) = h(n) + \mu[d(i) - y(i)]x^{*}(3i+n)$

The 1/3T Equalizer algorithm consists of three sections:

1. The first section computes the complex multiplication of the input data and the equalizer coefficients.
2. The second section formats the output *y(IQ)* and generates the error *term ((v(IQ) y(IQ)) >>4).*
3. The third section performs the adaptation and updates the equalizer coefficients *h(IQ).*

3.0. IMPLEMENTING A 1/3T EQUALIZER IN MMX™ INSTRUCTIONS

This 1/3T Equalizer, as implemented in this application note, is one functional block in a modem application. This means that the input and output data formats were predefined by the modem application. Each of the modem functions expect the input and output data in the predefined format. The predefined data formats were not optimal for MMX technology adaptation, so the data was reformatted. This reformatting step was done outside of the MMX technology routines, but it did take place within the equalizer function, so the reformatting does not disrupt the other functional blocks in the modem application. Once the data was reformatted, the MMX code optimization was straight forward.

There are two areas in this algorithm that worked especially well with MMX instructions. These were the two loops that perform a complex multiply of the equalizer input and coefficients, and the adaptation loop. This section will concentrate on the MMX code optimization and techniques used during this programming exercise.

3.1. Reformatting the Input and Output Data

The MMX code was written with the assumption that the data is formatted as follows. Each term is a 16-bit value.

The *real (sI)* and *imaginary (sQ)* terms of the data input are stored as *[sI: -sQ: sQ: sI]*.

The *real (hI)* and *imaginary h(Q*) terms of the filter coefficients are stored as *[hI: hQ]*.

The *real y(I)* and *imaginary y(Q)* terms of the output are stored as *[yI: yQ]*.

A C language code wrapper is shown in Appendix A. The function of this wrapper is to reformat the data for efficient implementation of the 1/3T Equalizer. The wrapper was created to preserve the data format for the rest of the algorithms in the modem application. The need for data reformatting is a function of the implementation environment for this 1/3T Equalizer function. It may not be a step that is always necessary.

3.2. Calculating the Filter Output Y(IQ)

InnerLoop 1 (see Appendix B) computes an intermediate sum used to calculate the real and imaginary terms of the equalizer output. A complex multiplication needs to occur between the equalizer inputs and the equalizer coefficients for the length of the equalizer coefficient array. A complex multiply can be performed easily in MMX instructions if the data is preformatted. In the case of the equalizer, the complex filter coefficients in the form *[0: 0: hI: hQ]* can be multiplied by a preformatted complex data input *[sI: -sQ: sQ: sI]*. Example 1 illustrates this idea.

Example 1. Preformatted Complex Data Input

```
PXOR            mm7, mm7                ; clear mm7 and use it to accumulate the
ccomplex result
MOVD            mm1, the_filter_coef    ; mm1=[0:0:hI:hQ]
PUNPCKLDQ       mm1, mm1                ; mm1=[hI:hQ:hI:hQ]
PMADDWD         mm1, the_data_input     ; mm1=[hIsI - hQsQ: hIsQ + hQsI]
PADDD           mm7, mm1                ; accumulate the result in mm7
```

The latency of the PMADDWD instruction is three clocks and the PMADDWD result is needed to do the PADDD. Rather than waste these potential execution slots, we decided to unroll the loop and do three complex multiplies in parallel. We chose the loop unrolling number by filling out the chart shown in Example 2 and working out, on paper, a scheme for pairing instructions. An effective way to begin pairing the instructions is to place one PMADDWD in the U pipe before the loop control instructions and work backward. The PMADDWD results are not available until the third clock after the instruction is executed and the pointer manipulation and loop control takes two clocks. This allows us to use the pointer manipulation and loop control instructions while waiting for the PMADDWD result.

Example 2. InnerLoop1 Optimization

```
        n       n+1     n+2
-----------------------------------------
        u       MOVD
        v       PADDD
        u       MOVD
        v       PUNPCKLDQ
        u       PMADDWD
        v       PADDD
        u       MOVD
        v       PUNPCKLDQ
        u       PMADDWD
        v       PUNPCKLDQ
        u       PMADDWD
        v       PADDD
                pointer management 1 clock
                loop control= 1 clock
```

The loop in Example 2 is unrolled three times (*n* represents loop unrolling). The key points are to place a PMADDWD instruction in the clock before the pointer management and loop control to make use of these two clocks. Be sure that the PMADDWD instruction (result available on the third clock) immediately follows the PUNPCKLDQ. Since the unpack must happen first, and the multiply-add instruction relies on the result of the unpack. The choice as to whether to use the U or V pipe was dictated by the fact that memory references can only use the U pipe.

Example 3 shows the MMX code to implement this InnerLoop. There is code before this loop to set up the pointers for the data and coefficients, to clear registers used within the loop, and to set up the outer loop counter and filter length pointers. The entire listing can be reviewed in Appendix B. Since the loop was unrolled three times, there are some instructions that handle the pointer manipulation and loop count.

Example 3. MMX™ Code Example for InnerLoop1

```
; This loop performs the filtering operation. It executes the
first complex multiplication necessary to compute the output ;y(IQ).
The sum is accumulated in mm7.
InnerLoop1:
MOVD              mm1, [ebx]        ; fetch first element of hIQ
PADDD             mm7, mm3          ; accumulate 2nd partial    ; product in mm7

MOVD              mm3, 4[ebx]       ; fetch second element of hIQ
PUNPCKLDQ mm1, mm1                  ; copy 1st hIQ into upper half       ; of mm1
PMADDWD           mm1, [eax]        ; complex multiply the first         ; set of sIQ and hIQ elements
PADDD             mm7, mm5          ; accumulate 3rd partial    ; product in mm7
MOVD              mm5, 8[ebx]       ; fetch the 3rd element of hIQ
PUNPCKLDQ mm3, mm3                  ; copy 2nd hIQ into upper half       ; of mm3
PMADDWD           mm3, 8[eax]       ; complex multiply the second        ; set of sIQ and hIQ elements
PUNPCKLDQ mm5, mm5                  ; copy 3rd hIQ into upper half       ; of mm5
PMADDWD           mm5, 16[eax]      ; complex multiply the third         ; set of sIQ and hIQ elements
```

```
PADDD               mm7, mm1            ; accumulate 1st partial     ; product in mm7
ADD                 eax, 24             ; update eax to point to next          ; sIQ
ADD                 ebx, 12             ; update ebx to point to next          ; hIQ
SUB                 si, 3               ; decrement loop count by three ; because of loop unrolling
JNZ
InnerLoop1                              ; end of InnerLoop1
```

3.3. Formatting the Equalizer Output and Generating the Error Term

The second section of code formats and stores the equalizer outputs and generates the error term. In InnerLoop1 the output is accumulated into MM7. The high 32 bits of MM7 have the real term of the sum. The imaginary term is in the low 32 bits of MM7. The real and imaginary terms get added to a rounding constant and shifted right by 14. Shifting right by 14 will keep the high 16-bits of the multiply and apply a gain of two. The real and imaginary terms of the complex multiply are now contained in bits 47-32 and 15-0 respectively. This output is reformatted to bits 32-16:15-0 in MM7, and stored as the new equalizer output.

An error term must be generated and formatted for the adaptation step of the algorithm. This is done by testing the equalizer output (real and imaginary) for a greater than or equal to zero value. If the output is greater than or equal to zero, an intermediate term *v(IQ)* is set to a positive bias, otherwise the *v(IQ)* is a negative bias. The output *y(IQ)* is subtracted from the *v(IQ)* term and shifted right by four to generate the error term *e(IQ)*. The error term is formatted in MM4 to be in the form *[eI: -eQ: -eI: eQ]* . The error term is ordered like this because the adaptation multiply needs the complex conjugate. Now the inputs for InnerLoop2 are in a form to take advantage of the complex multiply technique for the adaptation section. The multiply result in innerLoop2 will be in the *form [eIsI + eQsQ: eQsI - eIsQ]*.

3.4. Performing the Adaptation and Generating New Filter Coefficients

The last section of code performs the adaptation calculation and generates the new filter coefficients. The C Language code that describes this is shown in Example 4.

Example 4. Adaptation C Language Code

```
        // adaptation
        for (i=0; i<h_Leng; i++){
            SumI=e.I*(long)sI[3+j+i]+e.Q*(long)sQ[3+j+i];
            SumQ=e.Q*(long)sI[3+j+i]-e.I*(long)sQ[3+j+i];
            SumI=((SumI+0x4000)>>15)+hI[i];
            SumQ=((SumQ+0x4000)>>15)+hQ[i];
            hI[i]=SumI;
            hQ[i]=SumQ;
    }
```

InnerLoop2 computes and stores the new filter coefficients used for the next iteration of InnerLoop1. In this loop a complex multiplication is performed with the input data and the error term. The real and imaginary terms get added to the rounding constant and shifted right by 15. The real and imaginary terms of this intermediate sum are reformatted to be contained in bits 32-16:15-0 of the MMX register. Now this intermediate sum can be added, with saturation, to the filter coefficients *h(IQ)* to create the new (adapted) filter coefficients for the next iteration of the outer loop. The instruction set's saturation option provides a low cost way to avoid a data dependent overflow that can occur during the early iterations of the equalizer algorithm.

Example 5 shows the MMX code for InnerLoop2. It took 10 instructions to perform the operation. The loop was unrolled twice and used all available MMX registers. There were only two unused slots, a shift

and an unpack. Prior to entering the loop, the *s(IQ)* and *h(IQ)* pointers and the loop counter were set up. The complex multiply of the second *s(IQ)* element with the error term *e(IQ)* was started. This was done so we could enter InnerLoop2 with one PMADDWD already started.

Example 5. MMX™ Code Example for InnerLoop2

```
; This loop performs the adaptation operation.
It calculates the ; new filter coefficient terms.
InnerLoop2:
MOVQ            mm0, [eax]      ; fetch the 1st sIQ element
PADDD           mm2, mm5        ; add the rounding constant to
                                ; the 2nd complex product
MOVD            mm3, 4[ebx]     ; fetch the 2nd hIQ element
PMADDWD         mm0, mm4        ; complex multiply 1st sIQ with
                                ; the error term
MOVD            mm1, [ebx]      ; fetch the 1st hIQ element
PSRAD           mm2, 15         ; signed shift right 2nd partial
                                ; product by 15 bits
MOVQ            mm6, mm2        ; copy 2nd partial product to mm6
PADDD           mm0, mm5        ; add the rounding constant to the 1st complex product
PUNPCKHDQ       mm6, mm6        ; copy the real term of the 2nd
                                ; partial product to lower 32 bits of mm6
PSRAD           mm0, 15         ; signed shift right 1st partial
                                ; product by 15 bits
MOVQ            mm7, mm0        ; copy 1st partial product to mm7
PUNPCKLWD       mm2, mm6        ; format the real & imag. terms
                                ; of 2nd partial product in
                                ; [31..16] and bits[15..0]
                                ; respectively
PUNPCKHDQ       mm7, mm7        ; copy the real term of the 1st
                                ; partial product to lower 32
                                ; bits of mm6
PADDSW          mm3, mm2        ; perform saturated addition of
                                ; the 2nd partial product with hIQ

MOVD            4[ebx], mm3     ; store the (2nd)new filter
                                ; coeffs
PUNPCKLWD       mm0, mm7        ; format the real & imag. terms
                                ; 1st partial product in
                                ; bits[31..16] and bits[15..0]
                                ; respectively
MOVQ            mm2, 24[eax]    ; fetch the 2nd sIQ element
PADDSW          mm1, mm0        ; perform saturated addition of
                                ; the 1st partial product with
                                ; hIQ
Movd            [ebx], mm1      ; store the (1st)new filter
                                ; coeffs.
PMADDWD         mm2, mm4        ; complex multiply 2nd sIQ with
                                ; the error term
ADD             eax, 16         ; update eax to point to
                                ; next sIQ.
ADD             ebx, 8          ; update ebx to point to next hIQ
SUB             si, 2           ; decrement loop count by two
                                ; because of loop unrolling
JNZ             InnerLoop2      ; end of InnerLoop2
```

4.0. MMX™ TECHNOLOGY CODE PERFORMANCE

The equalizer function executed in 0.72 cycles per instruction. The instructions in the loops paired very well but the misaligned memory accesses caused memory stalls that prohibited further performance gains. Some further optimizations could be implemented if the instruction that caused memory stalls in both inner loops; MOVD mmx, 4[ebx] was replaced by PSRLQ mmx, 32. The instruction pairing would have to be done again but this could result in further clock saving.

APPENDIX A. C-CODE WRAPPER FOR 1/3T EQUALIZER

```
include <stdio.h>
/* This code is the c-wrapper around the optimized MMX code that will format
/* the input and output data in a way to take advantage of the MMX techniques */
extern void Equ13TAsm(short *sP, short *hP, short *yP, short h_Leng, short x_Leng);
void *AlignAlloc(unsigned int nbytes);
void Equilizer13MMx(short *sP, short *hI, short *hQ, short *xI, short *xQ, short *yI,
short *yQ, short h_Leng, short x_Leng)
{
      short i, k ;
                  short *hP, *yP ;
                  for (i=0; i< x_Leng; i++) {
                        k = (h_Leng+i)*4 ;
                        sP[k]   =  xI[i];
                        sP[k+1] =  xQ[i];
                        sP[k+2] = -xQ[i];
                        sP[k+3] =  xI[i];
                  }
/* Allocate the space for interleaving hI and hQ.
   The hI/hQ pair will be organized as: Real Imag.
   Unlike the sI/sQ pair, the data will not be duplicated. */
                  hP = (short *)AlignAlloc(h_Leng*4);

                 for (i=0; i< h_Leng; i++) {
                       hP[2*i]   = hQ[i];
                       hP[2*i+1] = hI[i];
                  }
/* Allocate the space for interleaving yI and yQ.
   The yI and yQ output by Equ13T will be organized as: Real Imag. */
                       yP = (short *)AlignAlloc(x_Leng*4);
/* Call the MMX routine */
                       Equ13TAsm(sP, hP, yP, h_Leng, x_Leng);
/* update initial state */
                      for (i=0; i<h_Leng; i++) {
                            k = (x_Leng+i)*4;
                            sP[4*i]   = sP[k];
                            sP[4*i+1] = sP[k+1];
                            sP[4*i+2] = sP[k+2];
                            sP[4*i+3] = sP[k+3];
                      }
/* Update yI and yQ as modified by Equ13T */
                      for (i=0; i<x_Leng; i=i+3) {
                           yI[i/3] = yP[(i/3)*2+1];
                           yQ[i/3]= yP[(i/3)*2];
                     }
/* Update hI and hQ as modified by Equ13T */
                      for (i=0; i<h_Leng; i++) {

                            hQ[i] = hP[2*i];
                            hI[i] = hP[2*i+1];
                      }
 }
 void *AlignAlloc( unsigned int nbytes)
 {
       char *cptr;
       cptr = (char *)malloc(nbytes+8);
        if (!cptr)
       {
            perror("xalloc: Error allocating memory");
            exit(1);
        }
```

```
        cptr = (char *)((unsigned long)cptr & 0XFFFFFFF8);
        return(cptr);
}
```

APPENDIX B. MMX™ CODE FOR 1/3T EQUALIZER

```
INCLUDE iammx.inc
TITLE fil1t3
.486P
.model FLAT
_DATA SEGMENT
; Define all the constants/local variables
; const_rnd:     to store the rounding value
; pos_bias:       to store the positive bias for clipping
; neg_bias:       to store the negative bias for clipping
; err_fmt_one:  pattern "ffff0000 0000ffff" used while formatting the error value
; err_fmt_two:  pattern "0000ffff ffff0000" used while formatting the error value
const_rnd     DWORD  4000H, 4000H
pos_bias      DWORD         08000800H
neg_bias      DWORD   0f800f800H
err_fmt_one   DWORD   0000ffffH, 0ffff0000H
err_fmt_two   DWORD   0ffff0000H, 0000ffffH
_DATA ENDS
; ****************** ASSUMPTIONS *****************
; The real(sI) & imag.(sQ) terms of the data input are stored as:
;bits->63..            ..0;
;        sI : -sQ : sQ : sI
; Before the second inner loop(adaptation operation), the real(eI)
; and imag.(eQ) components of the error term will be formatted as:
;bits->63..             ..0
;          eI : -eQ : -eQ : eI
; The real(hI) & imag.(hQ) terms of the filter coeff. are stored as:
;bits->   31....0
;             hI : hQ
; The real(yI) & imag.(yQ) terms of the output are stored as:
;bits->  31....0
;            yI : yQ
; *********************************************
_TEXT SEGMENT
; setup the pointers for sI/sQ, hI/hQ, yI/yQ.
; also set the pointers for storing the input length
; and filter coefficient length.
_sPtr$ = 28
_hPtr$ = 32
_yPtr$ = 36
_hLen$ = 40
_xLen$ = 44
_Equ13TAsm PROC NEAR USES ebx ecx edx ebp esi edi
 xor          edi, edi
 xor          esi,esi
; store sI/sQ pointer in ecx
; store hI/hQ pointer in edx
; store yI/yQ pointer in ebp
; store the input data length in di
; store the filter coeff length in loop_count
MOV           ecx, _sPtr$[esp]
MOV           edx, _hPtr$[esp]
MOV           ebp, _yPtr$[esp]
MOV           di, _xLen$[esp]
; This loop embeds two inner loops. The first inner loop performs
; the filter operation. The second inner loop performs the adaptation
; of the filter coefficients. Between the two inner loops, there is a
; scalar section of code that calculates the output and the error terms.
outerLoop:
       MOV          ebx, edx            ; copy hIQ ptr into ebx
```

```
        PXOR        mm7, mm7            ; initialize mm7 to zeros. One of
                                        ; the partial sums in InnerLoop1
                                        ; will be stored in mm7
        MOV         si, _hLen$[esp      copy filter length to si
                    his will be the loop counter
                                        ; for InnerLoop1
                                        ; Note that this assumes no stack
                                        ; pushes are done within this code
                                        ; If needed, this can easily be
                                        ; avoided by storing this value in a
                                        ; local variable
        PXOR        mm5, mm5            ; initialize mm5 to zeros
                                        ; mm5 will be used in InnerLoop1
                                        ; to store one of the partial
                                        ; results of complex multiply

        ADD         ecx, 24             ; initialize ecx to point to first
                                        ; element of sIQ for next iteration
                                        ; of InnerLoop1.
        MOV         eax, ecx            ; copy sIQ pointer to eax. eax will
                                        ; be used within InnerLoop1 to
                                        ; point to subsequent elements of sIQ
        PXOR        mm3, mm3            ; initialize mm3 to zeros. One of
                                        ; the partial sums in InnerLoop1
                                        ; will be stored in mm3
                                        ; mm1 will also be used in storing
                                        ; the partial sum. But it does not
                                        ; have to be initialized to zeros
                                        ; since it is used only towards the
                                        ; end of the loop(after calculating
                                        ; the partial product).
; This loop performs the filtering operation. It executes the first
; complex multiplication necessary to compute the output y(IQ)
innerLoop1:
        MOVD        mm1, [ebx]          ; fetch first element of hIQ
        PADDD       mm7, mm3            ; accumulate 2nd partial product in mm7
        MOVD        mm3, 4[ebx]         ; fetch second element of hIQ
        PUNPCKLDQ   mm1, mm1            ; copy 1st hIQ into upper half of mm1
        PMADDWD     mm1, [eax]          ; complex multiply the first set of
                                        ; sIQ and hIQ elements
        PADDD       mm7, mm5            ; accumulate 3rd partial product in mm7
        MOVD        mm5, 8[ebx]         ; fetch the 3rd element of hIQ
        PUNPCKLDQ   mm3, mm3            ; copy 2nd hIQ into the upper half of mm3
        PMADDWD           mm3, 8[eax]         ; complex multiply the second set of
                                        ; sIQ and hIQ elements
        PUNPCKLDQ    mm5, mm5           ; copy 3rd hIQ into upper half of mm5
        PMADDWD     mm5, 16[eax]              ; complex multiply the third set of
                                        ; sIQ and hIQ elements
        PADDD       mm7, mm1            ; accumulate 1st partial product in mm7
        ADD         eax, 24                   ; update eax to point to next sIQ
        ADD         ebx, 12             ; update ebx to point to next hIQ
        SUB         si, 3               ; decrement loop count by three
                                        ; because of loop unrolling
        JNZ         InnerLoop1          ; end of InnerLoop1
;This next section of code formats the output term (most sig 16 bits) with a gain ;f 2.
;Also, generate the error term and format it to take advantage of the complex ;multiply
technique.
        MOVQ        mm6, const_rnd            ; fetch the rounding constant
        PADDD       mm7, mm3            ; accumulate the partial product
                                        ; from the iteration of InnerLoop1
        MOVD        mm2, pos_bias       ; fetch the positive bias for
                                        ; clipping to generate the error
        PADDD       mm7, mm5            ; accumulate the 3rd partial product
                                        ; from InnerLoop1
```

```
        MOVD        mm3, neg_bias       ; fetch the negative bias for clipping to
                                        ; generate the error
        PADDD       mm7, mm6            ; add the rounding constant to the
                                        ; InnerLoop1 product
        PSRAD       mm7, 14             ; signed shift the real and imaginary sums by 14 to
                                        ; keep the high 16 bits of the multiply and
                                        ; comprehend a gain of 2. The real and imaginary
                                        ; terms of the output are now in bits 0..15,
                                        ; 32..47.
        PXOR        mm0, mm0            ; initialize mm0 to zero
        MOVQ        mm6, mm7            ; copy yIQ output to mm6
        PXOR        mm1, mm1            ; initialize mm1 to zeros.

        MOVQ        err_fmt_one         ; move ffff00000000ffff to mm5
        PUNPCKHD    Qmm6, mm6           ; mm6 now contains yI in bits 0..15 & 32..47
                                        ; bits 16..31 &48..63 are 1.
        PUNPCKLWD   mm7, mm6            ; yQ is in bits 0..15, yI is in bits 16..31
        MOVQ        mm4, mm7            ; copy yIQ into mm4
        PCMPEQW           mm0, mm7          ; check the real and imag. terms of output
                                        ; for zero equality
        MOVDf       [ebp], mm7          ; store the yIQ output
        PCMPGTW           mm4, mm1          ; check the real & imag. terms of output
                                        ; for > zero
        ADD         ebp, 4              ; update ebp to point to next output
        POR         mm0, mm4            ; or the results of the > and = checks.
                                        ; create a bit pattern to use for setting the
                                        ; error term bias(vIQ) to pos. or neg.
        PAND        mm2, mm0            ; store positive bias in mm2 bits[31..0]
                                        ; if yI >= 0 and yQ >= 0
        PANDN       mm0, mm3            ; store negative bias in mm0 bits[31..0]
                                        ; if yI < 0 and yQ < 0

        POR         mm0, mm2            ; compute vI and vQ terms, used in
                                        ; calculating the error terms.
        PXOR        mm4, mm4            ; initialize mm4 to zeros
        MOVQ        mm1, err_fmt_two    ; mm1 has 0000ffffffff0000
        PSUBW       mm0, mm7            ;compute the diff. between vIQ and yIQ
        MOV         eax, ecx            ; copy sIQ pointer to eax. eax will
                                        ; be used within InnerLoop2 to
                                        ; point to subsequent elements of sIQ
        PSRAW       mm0, 4                  ; signed shift right mm0 by 4    bits
                                        ; the real & imag. terms of the error
                                        ; are in bits [31..16] and bits[15..0]
                                        ; respectively
        MOVQ        mm2, 8[eax]         ; fetch the 2nd sIQ element for complex
                                        ; multiplication in InnerLoop2. The 1st
                                        ; sIQ element will be fetched within
                                        ; InnerLoop2. In this case, the 2nd
                                        ; partial product is being computed
                                        ; before the first one for better pairing
        PUNPCKLDQ   mm0, mm0            ; copy real & imag. error terms to
                                        ; upper half...bits[63..32]
        MOV         ebx, edx            ; copy hIQ ptr into ebx. will be used
                                        ; within InnerLoop2 to point to
                                        ; subsequent elements of hIQ
        PSUBW       mm4, mm0            ; compute the negative of real & imag.
                                        ; error terms in mm4; -eI:-eQ:-eI:-eQ
        PAND        mm0, mm5            ; format real(eI) & imag(eQ) error terms in
                                        ; bits[63..48] & bits[15..0]
        PAND        mm4, mm1            ; format -eQ and -eI in bits[47..32]
                                        ; and bits[31..16] respectively
        MOVQ        mm5, const_rnd          ; fetch the rounding constant
        POR         mm4, mm0            ; get eIQ in the form: eI:-eQ:-eI:eQ
        MOV         si, _hLen$[esp]     ; copy filter length to si
```

```
                                        ; this will be the loop counter
                                        ; for the InnerLoop2
                                        ; Note that this assumes no stack
                                        ; pushes are done within this code
      PMADDWD           mm2, mm4            ; compute the partial product from
                                        ; complex multiplication of the 2nd
                                        ; term of sIQ and eIQ
; This loop performs the adaptation operation. It calculates the new
; filter coefficient terms.
innerLoop2:
      MOVQ        1mm0, [eax]         ; fetch the 1st sIQ element
      PADDD       mm2, mm5            ; add the rounding constant to
                                        ; the 2nd complex product
      MOVD        mm3, 4[ebx]         ; fetch the 2nd hIQ element
      PMADDWD           mm0, mm4            ; complex multiply 1st sIQ with
                                        ; the error term
      MOVD        mm1, [ebx]          ; fetch the 1st hIQ element
      PSRAD       mm2, 15                   ; signed shift right 2nd partial
                                        ; product by 15 bits
      MOVQ        mm6, mm2            ; copy 2nd partial product to mm6
      PADDD       mm0, mm5            ; add the rounding constant to
                                        ; the 1st complex product
      PUNPCKHDQ   mm6, mm6            ; copy the real term of the 2nd partial
                                        ; product to lower 32 bits of mm6
      PSRAD       mm0, 15                   ; signed shift right 1st partial
                                        ; product by 15 bits

      MOVQ        mm7, mm0            ; copy 1st partial product to mm7
      PUNPCKLWD   mm2, mm6            ; format the real & imag. terms of
                                        ; 2nd partial product in bits[31..16]
                                        ; and bits[15..0] respectively
      PUNPCKHDQ   mm7, mm7            ; copy the real term of the 1st partial
                                        ; product to lower 32 bits of mm6
      PADDSW      mm3, mm2            ; perform saturated addition of the 2nd
                                        ; partial product with hIQ
      MOVDf       4[ebx], mm3         ; store the (2nd)new filter coeffs.
      PUNPCKLWD   mm0, mm7            ; format the real & imag. terms of
                                        ; 1st partial product in bits[31..16]
                                        ; and bits[15..0] respectively
      MOVQ        mm2, 24[eax]        ; fetch the 2nd sIQ element
      PADDSW      mm1, mm0            ; perform saturated addition of the 1st
                                        ; partial product with hIQ
      MOVDf       [ebx],mm1           ; store the (1st)new filter coeffs.
      PMADDWD     mm2, mm4            ; complex multiply 2nd sIQ with
                                        ; the error term
      ADD         eax, 16                   ; partially update eax to point to
                                        ; next sIQ.
      ADD         ebx, 8              ; update ebx to point to next hIQ
      SUB         si,                 ; decrement loop count by two because
                                        ; of loop unrolling
      JNZ         InnerLoop2          ; end of InnerLoop2
      SUB         di, 3               ; decrement outerloop count by 3
      JNZ         outerLoop           ; end of outerloop
Done:
      emm                             ; flush FP stack
ret 0
_Equ13TAsm ENDP
_TEXT  ENDS
END
```